PAPER VHDL vs. SystemC: Design of Highly Parameterizable Artificial Neural Networks

David ALEDO^{†a)}, Member, Benjamin CARRION SCHAFER^{††}, and Félix MORENO[†], Nonmembers

SUMMARY This paper describes the advantages and disadvantages observed when describing complex parameterizable Artificial Neural Networks (ANNs) at the behavioral level using SystemC and at the Register Transfer Level (RTL) using VHDL. ANNs are complex to parameterize because they have a configurable number of layers, and each one of them has a unique configuration. This kind of structure makes ANNs, a priori, challenging to parameterize using Hardware Description Languages (HDL). Thus, it seems intuitively that ANNs would benefit from the raise in level of abstraction from RTL to behavioral level. This paper presents the results of implementing an ANN using both levels of abstractions. Results surprisingly show that VHDL leads to better results and allows a much higher degree of parameterization than SystemC. The implementation of these parameterizable ANNs are made open source and are freely available online. Finally, at the end of the paper we make some recommendation for future HLS tools to improve their parameterization capabilities.

key words: VHDL, SystemC, high-level synthesis (HLS), artificial neural network (ANN)

1. Introduction

With the increase time to market pressure, companies are relying on third party IPs (3PIPs), which are often highly parameterizable to create their complex digital circuits. They are also dedicating much effort to develop modules in-house which can be re-used in future projects. Artificial Neural Networks (ANNs) do not escape this trend. Thus, it makes sense to have highly parameterizable ANN descriptions that can be used in multiple different applications. Some parameters include, the number of layers, number of neurons and activation function.

ANNs have gained tremendous importance recently. They belong to the group of artificial intelligence (AI) because they have the capability of learning. Their name comes from the fact that they are biologically inspired on a simple model of a neuron, connected in a network similar to the brain ones. ANNs have an enormous amount of parallelism, making them excellent candidates for hardware (HW) acceleration. This allows the full exploitation of this parallelism while being extremely energy efficient compared to using general purpose solutions. The latest A11 Bionic application processor or Apple is one example, which includes a dedicated hardware module for the ANN. Typical speed-ups between 30-200x over SW implementations have been reported [1].

Traditional Hardware Description Languages (HDLs) like VHDL and Verilog, provide mechanisms to parameterize descriptions. One most notorious examples includes the use of *generics* in VHDL. Nevertheless, one additional trend that is happening to further speed-up the design of ICs, is to raise the level of abstraction from Register Transfer Level (RTL) to behavioral level. This implies that software descriptions initially intended to be compiled on a fixed architecture are synthesized through High-Level Synthesis (HLS) into an RTL description that can effectively execute it. All modern commercial HLS tools can take SystemC as input. SystemC is a C++ class that allows to model concurrency and has been standardize by the IEEE [2], [3].

Raising the level of abstraction has numerous advantages. One that can be applied to ANNs is that it allows, in theory, a much wider set of parameterization compared to low-level HDLs. Thus, using C-based VLSI design should lead to the designer of easier and more efficient ANNs. Whereas Ad-Hoc optimizations are described in a low abstraction level in order to take advantage of low-level details. Hence, these descriptions are difficult to reuse. The high-level descriptions are more general, therefore they are more likely to be reused.

This work presents our experience creating highly parameterizable ANNs using VHDL and SystemC. Both levels of abstractions will be compared and the different configuration parameters exposed. Finally we make some recommendations of how to enhance the way HLS tools parameterize behavioral descriptions for synthesis. In particular the contributions of this work can be summarized as:

- Study the key parameters that allow the description of highly parametrizable, and thus, re-usable, ANNs in SystemC and VHDL.
- Present experimental results, which quantify the effect of these parameters on the number of ANN configurations and quality of the ANNs.
- Propose suggestions when designing at behavioral level to improve configurability.
- Open source release of the ANNs described in SystemC and RTL.

It should be noted that the main aim of this work is not to develop a specific ANN, but to understand the trade-offs and limitations between the different levels of abstractions

Manuscript received April 19, 2018.

Manuscript revised September 21, 2018.

Manuscript publicized November 29, 2018.

 $^{^{\}dagger} The$ authors are with the CEI, Universidad Politécnica de Madrid, Spain.

^{††}The author is with Department of Electrical and Computer Engineering, University of Texas at Dallas, USA.

a) E-mail: david.aledo@upm.es

DOI: 10.1587/transinf.2018EDP7142

when designing complex IPs like ANNs.

The contents of this paper are organized as follows. First, a summary of related work on HW ANNs is given in Sect. 2. Then, background information about ANNs is given in Sect. 3. Then, the implementation at the RTL (VHDL) is shown on Sect. 4. Basic knowledge of VHDL is expected by the reader. Nevertheless, advanced and key points of VHDL are explained. Section 5 is divided on three parts. The first, introduces main concepts of HLS, the second one describes SystemC, and the third one shows the ANN description in SystemC. The next section compares both approaches and analysis the two methodologies in Sect. 6. The last section summarizes and concludes the presented work.

2. Related Work

Although the ANN implementation into an FPGA is used in this paper as an application example of a complex parameterizable design for the methodology and language analysis, it is still a hot-topic. A large variety of ANN hardware implementations have been published targeting FP-GAs, ASICs, or both [4]–[6]. In most of this previous work, the ANN is described at the RT-level, using either VHDL or Verilog. More recent work use HLS to construct the ANNs. Some examples include [7]–[14], of which [15], [16] make use of SystemC.

The contribution of this paper in not only the improved level of configurability of the ANN, but also the analysis and comparison of both design methodologies (Behavioral vs. RTL); and more important, the points of improvement for future HLS tools, as a conclusion of this analysis. Achieving a good level of configurability at the RTL, as will be shown in Sect. 4, is not complicated. A good example is [17]. In this previous work, the authorts parameterize an ANN in a similar way using generics and generate statements. However, our VHDL IP includes three kinds of layers in order to make it even more flexible in terms of area vs. performance trade-off. In [18], the authors also use a parameterizable VHDL IP to perform a DSE with their own tool.

In addition in [19], the authors have a parameterizable SystemC-TML code for DSE and simulation, however they have to convert it to synthesizable SystemC for the selected configuration. In [20] successfully parameterize layers for HLS using templates, however they need to instantiate each layer separately with their own scalar parameters, *i.e.* the number of layers is not a parameter. So, the code of the IP must be modified if a different number of layers is needed. Another solution is to have a tool that parses an even higherlevel description, such as graphical based one or using Caffe like the re-Vision stack from Xilinx [21]. This approach is useful when the tool is also used for training the ANN or carrying an automated DSE like in our previous exploration work [22]. This last approach is gaining popularity as it fully abstracts the details of the ANN away from the user. Other examples include [23]–[27], which propose easy to use frameworks to generate ANNs. Nevertheless, to have a parameterizable ready-to-use IP is convenient. For example, for inserting it directly on the FPGA programming tool like Vivado without needing to call an external tool to rewrite the IP code every time that it is reconfigured. Besides, it allows to make changes or add new functionality directly into the source code, and reuse the code in other projects. On the other hand, frameworks obscure the source code or make it inaccessible.

In summary, much work in the area of hardware implementations of ANNs is currently being done. We believe that this work is complementary to this previous work as it not only describes in detailed how to design highly flexible ANNs at the behavioral and RT-Level, but also compares both approaches and finally exposes some of the limitations in current methodologies, making suggestions on how to relieve these.

3. Artificial Neuronal Network (ANN) Background

The mathematical model of a neuron is shown in Eq. (1). This model performs a weighted sum of the inputs and then applies an activation function to the output. The activation function is usually a linear function or a sort of saturation function like sigmoid functions.

$$y_i = f\left(\sum_{j=1}^M w_{ij} \cdot x_j + b_i\right), \tag{1}$$

with i = 1, ..., N, where x_j are the inputs, w_{ij} the weights, b_i the bias, f() the activation function, y_i the outputs, M the number of inputs, and N the number of neurons.

One neuron alone can do very few things, but combined together, they can perform complicated tasks. As in a brain, neurons are grouped by layers and connected in a way that the outputs of a layer become inputs for the next layer. Certain kind of ANNs called Recurrent Neural Networks (RNN) allow feedback connections, although both of the implementations described on this paper only allow feedforward connections. Every neuron in a layer uses the same activation function. Figure 1 shows a general feedforward ANN architecture. The small nodes represent the inputs of the network, and the large ones perform the Eq. (1).

The last layer is called output layer, since is the one which generates the network's outputs. The other layers are usually called hidden layers.

ANNs need to be trained. During the training phase, input examples are presented to the network, and a learning algorithm updates the neuron weights in order to complete the task or improve results. Regarding the information provided along the training examples, the ANN learning can be



Fig. 1 General 2 layer feedforward ANN.

classified as supervised, unsupervised, or reinforcement.

For supervised learning, the desired outputs of each training input example are needed. This enables the learning algorithm to compare the desired outputs with the outputs obtained outputs from the ANN, and adjust the weights accordingly.

For the case of unsupervised learning, no information at all is provided with the input examples. The weights are updated following some rules that depend on internal characteristics of the input data. A typical application of this type of learning mechanisms is clustering, and the most representative ANNs of this kind of ANN are the Self-Organizing Maps (SOMs).

Finally, reinforcement learning does also not have the desired outputs for an input example, but only a measure of how good or bad the results are.

There are different families/types of ANNs. Each family has different connections, activation functions, and training methods. The ANN family targeted in this work is a multi-layer perceptron (MLP), which uses supervised training and only feedforward connections. On the context of deep learning, which has gain grater popularity in the last years, this kind of ANNs are called fully connected layers and they are the last layers of a Convolutional Neural Network (CNN). To achieve a full CNN, convolutional and pooling layers should be added. These layers have similar characteristics than the MLPs and hence, the same conclusions apply to them.

Backpropagation [28] is the most popular supervised learning algorithm for MLP. It is an approximate steepest descent algorithm which uses the mean square error (MSE) between the ANN output and the desired one as the fitness function. Where $MSE = \frac{1}{n} \sum_{i=1}^{n} (t_i - y_i)^2$, t_i is the desired output (target vector), and y_i is the current ANN output. The neurons' weights are updated using the following formula:

$$\Delta w_{ij} = -\alpha \cdot \frac{\partial e^2}{\partial w_{ij}} , \qquad (2)$$

where e^2 is the MSE, and $\alpha \in [0, 1]$ is a scalar coefficient called learning rate which regulates how fast the ANN learns.

In this analysis, we omit the ANN training part as this is an iterative process and focus on the implementation of the ANN.

4. RTL Implementation with VHDL

The objective of this work is to describe an ANN as general as possible in order to be able to reuse it on different applications. From the ANN algorithm described on Sect. 3, the following parameters, listed on Table 1, have been identified as the main parameters that allow the parameterizations of the ANN. It should be noted that the ANN described in this section has been made fully available on OpenCores.org [29].

These parameters can be categorize in two main groups: The first affect the complete ANN and include Nlayer, NbitW, NumIn, NbitIn, and NbitOut. Hence, they

Table 1List of the VHDL ANN IP generics.

Parameter	Туре	Description
Nlayer	integer	Number of layers
NbitW	natural ^a	Bit width of weights and biases
NumIn	natural	Number of inputs to the network
NbitIn	natural	Bit width of the inputs
NumN	int_vector ^b	Number of neurons in each layer
1_type	string ^c	Layer type of each layer
f_type	string ^d	Activation function type of each layer
LSbit	int_vector ^b	LSB of the output of each layer
NbitO	int_vector ^b	Bit width of the outputs of each layer
NbitOut	natural	Bit width of the network output

^a NbitW should be a multiple of 8 due to memory alignment.

^b int_vector is an array of integers defined on the package layers_pkg.vhd.

^c Ltype string must contain as layer types as number of layers, separated by spaces. Each layer type is a two character string "SP", "PS", or "PP".

^d f_type string must contain as activation function types as number of layers, separated by spaces. Each layer type is a six character string.

are scalars. The second group affect each layer separately and include NumN, l_type, f_type, LSbit, and NbitO. Thus, they are vectors. NbitW, NbitIn, Lsbit, NbitO, and NbitOut configure fixed point data formats, while the other parameters configure the ANN's structure and functionality.

As shown from Fig. 1, the main components of an ANN are the layers and neurons. Thus, the main effort is in parameterizing the RTL description, such that each layer can be uniquely configured through static parameters instead of having to specify each layer and within each layer, each neuron, separately. This can be achieved in VHDL using *generics*, which are passed to the entity when instantiated. These *generics* are in turn used in *for-generate* statements in the architecture to generate the particular ANN layer.

Figure 2 shows a VHDL code snippet using a forgenerate statement for instantiating one module *Number_of_instances* times. The loop counter of the for-generate (*i*) statement ("**for** i **in** 0 **to** 2 **generate**") is an integer constant named *i* with values 0 to *Number_of_instances*. Every iteration is concurrent and thus, is equivalent to copy and paste the code for each iteration replacing *i* by the appropriate loop iteration value.

Thanks to the generics, the constant array types, and the fact that the for-generate loop counters are constant, each entity instance can be configured separately. In Fig. 2 for example, every instance of the entity *module* is configured with the same *global_parameter* and is connected to the same *clk* signal. But, each one is configured with a different element of the *array_parameter* and is connected to a different element of the *array_signal*.

Another interesting feature of VHDL and how VHDL descriptions are synthesized is the calculation of constants. In order to optimize HW designs, every value known at synthesis time can save all the logic needed for its computation. Results of operations with constants are treated as constants

Fig.2 VHDL for-generate statement.

as well, even when functions are applied (as long as this functions do not use any run-time values). As opposite to SW compilers, which leave these computations to when the code is executed[†]. This is mainly the case to increase the flexibility in SW albeit a slighter larger object codes.

One of the main drawbacks of RTL synthesis against HLS, is the fixed architecture. On RTL abstraction every clock cycle data calculation or movement is detailed. Therefore, trade offs between area and performance have to been decided *a priori* and coded explicitly. This means that have to be decided which calculations will be parallelized and which ones will be serialized to reuse resources. If a future application needs another trade off, old code can not be reused and has to be rewritten.

In order to give some flexibility on the ANN area versus performance trade off, three kinds of layers have been designed: Serial-input Parallel-output ("SP"), Parallel-input Serial-output ("PS"), and Parallel-input Parallel-output ("PP").

- The "SP" layer type is an array of Multiplier-and-Accumulators (MACs). Its resource utilization depends on the number of neurons, and its latency depends on the number of inputs. This is the most common implementation and is perfect for first processing layers which receive inputs serially.
- The "PS" layer type implements one neuron that is reused to calculate all neurons of the layer. Its resource utilization depends on the number of inputs, and its latency depends on the number of neurons plus the logarithm to the base 2 of the number of inputs (because the adder tree). A drawback of this layer type is that there is no perfect mapping of the multipliers and the adder tree into embedded MACs like Xilinx's DSP48Es. They have one multiplier and one adder, configurable to perform different operations like MAC, but to combine all multipliers with the adder tree adders is not possible, causing more DSPs utilization. Never-



Fig. 3 Block diagram of the architecture around one layer. This is repeated Nlayer times.

theless, this layer type is good for output layers which need serial outputs.

• The "PP" layer type is the full parallel implementation of a layer, achieving the maximum performance at expenses of the largest resource utilization.

When serial-input and serial-output is needed, this can be automatically accomplished by inserting a serializer after a "SP" layer or a parallelizer before a "PS" layer. A serializer is a shift register with parallel load, and a parallelizer is a shift register with parallel unload.

All neurons in a layer have the same activation function. The selected activation function is inserted selectively between layers. If a parallel output precedes a serial input, a serializer is inserted before a single activation function block. If a serial output precedes a parallel input, one activation function block is inserted before the parallelizer. Only when a parallel output precedes a parallel input, an array of parallel activation function blocks is inserted. Figure 3 shows the configurable architecture around one layer.

This high configurable architecture is achieved by making use of VHDL's for-generate and if-generate statements. However, programming this degree of flexibility requires a large designer effort and it still has its limitations (each option has to be coded as a different submodule). Reaching the point where to write a new design for a new option is easier than making a more complex description.

5. HLS with SystemC

5.1 High-Level Synthesis

HLS takes an untimed behavioral description, and trans-

[†]This is true for earlier versions of C++ than C++11. However, even in C++11 and newer versions, static time constant calculation must be specified on the code through the key word "constexpr". Thus, already existing functions (like library ones) can not be used for this propose.



Fig. 4 For loop and its rolled (left), partial unrolled (middle), and unrolled (right) implementations.

forms it in an efficient RTL description that can execute it [30]. The main steps of HLS are allocation, scheduling and binding. In particular, allocation specifies the hardware resources that are necessary to implement the particular description. Scheduling determines for each operation the time at which it should be performed such that no precedence constraint is violated. Binding provide a mapping from each operation to a specific functional unit and from each variable to a register.

HLS is intended to reduce the productivity gap between algorithm specification and HW design. Traditional HW design is usually perform at RTL with HDLs like VHDL and Verilog, whereas algorithm specifications are usually done with high-level languages like C, C++, Matlab or Python. So, manual translation is needed, which is a timeconsuming and error-prone process. HLS partially automatizes this process. It is not yet a silicon compiler who transforms whatever algorithm into a chip without manual refinement, although it almost significantly reduces this productivity gap. In particular, guidance through the optimizations are needed, and translation from standard C/C++ algorithm descriptions into synthesizable C like descriptions is still needed. Nevertheless, this translation is more straightforward than translating C/C++ to VHDL or Verilog, thus easier, faster, and less error prone.

Furthermore, HLS can synthesize one description into different RTL architectures. E.g. two of the main features of HLS to obtain different architectures are array mapping and loop unrolling (Fig. 4). Arrays can be synthesized as RAM/ROM memory blocks, register benches, or even as shift registers or FIFOs. If there are no data dependencies between loop iterations, a loop can be unrolled to achieve a parallel implementation with maximum performance but larger area. Alternatively, it can be kept sequential to allow reuse of the hardware described inside the loop body, achieving minimum area, with the obvious trade-off of larger latencies. Besides, it can be partially unrolled by instantiating different level of parallel copies of the loop's hardware, and reusing them serially to complete the remaining loop iterations. These synthesis knobs enable the generation of designs with unique trade-offs, e.g. area vs. performance without the need to modify the behavioral description at all.

Thus, HLS enables to perform Design Space Exploration (DSE), and allows its automation.

First generations of HLS tools used HDL as input

sources, but they were a commercial failure due to the reasons explained in [31]. Present HLS tools uses C/C++ based languages, however from all of them, only SystemC is standard for HLS through the synthesizable subset [3]. Although HLS tools accepts ANSI C++, like Vivado and CatapultC, their synthesizable subsets and some C++ translations into HW (data types, interfaces,...) are not standard. Thus, only SystemC is portable among HLS tools. Thus, this work focuses on implementation of ANNs using SystemC as input language for HLS.

5.2 SystemC

SystemC is a C++ library to model hardware. It has two main advantages compared to ANSI-C/C++. First it allows to model concurrency through modules, threads and methods and secondly it contains standard data types (integer and fixed point).

The first version of SystemC was just another RTL language. It defined the simulation kernel, modules, signals and fixed point data types. However, as RTL languages VHDL and Verilog are more powerful. Nevertheless, current version of SystemC includes higher-level abstraction elements which make SystemC capable of targetting a wide range of abstraction levels. This is very useful for simulation and validation.

Although all the C++ and SystemC features can be used for simulations and validation, Accellera has defined a synthesizable subset for HLS that restricts the supported syntax [3]. *E.g.* dynamic memory allocation is not allowed, as well as recursion or floating point number representations.

SystemC descriptions has a slight lower abstraction level than C++ descriptions for HLS (at least than the Vivados's ones). C/C++ are truly untimed algorithm descriptions. No concurrency is explicitly described, so synchronization is not needed. Data dependencies and concurrency is extracted by the tool, thus designer rely on the tool and its C/C++ extensions. SystemC allows designers explicitly describe concurrency and synchronization, giving more control on HLS at expenses of reducing slightly the abstraction level. This has its advantages and drawbacks.

When abstraction level increases code reusability becomes more important, as has been point out on the introduction. Furthermore, since high-level abstraction descriptions do not *lock* the micro-architectures, they can be more easily reused to generate micro-architectures of unique area vs. performance trade-off.

However, configurability is a weak point of SystemC. Due to the fact that SystemC is a C++ library, its parameterization mechanisms are defined by the ANSI C++ standard. Although it might seem counterintuitive, because C++ is well known as one of the most flexible languages, the C++ standard was created for software development. Although SystemC is a good adaptation of a software language to a hardware description language, which has successfully addressed most of these differences, parameterization still has some issues related with this fact which becomes visible when designing ANNs with SystemC.

The main parameters that can be used to parameterize a module in SystemC are: macros, templates, and constructor parameters. In particular:

- Macros: The #define preprocessor directive can define scalar global parameters. An advantage of macro parameters is they can be used in conjunction with #if preprocessor directives. Like VHDL if-generate statements, #if can be used to skip from synthesis those parts of the code which may produce errors for the selected configuration.
- **Templates:** the C++ template parameters can be configured independently for different instances of the same module. Thus, it seems that C++ templates are the best option to parameterize modules like VHDL generics do. However they have some limitations explained below.
- **Constructor parameters:** as SystemC modules are classes, they have constructor functions which may have parameters. The problem of constructor parameters (as any function parameters) is that from the C++ point of view they are variables known at run time. For this reason, constructor parameters are not synthesizable [3].

The main drawback of the C++ templates is they cannot interact with the preprocessor. They define parameters known at compiler time that can be used as constants and to perform some compiler-time optimizations, but they cannot be used to take decisions at synthesis time. This implies that C/C++ macros based on #if preprocessor statements are needed, like VHDL generics are used on if-generate and forgenerate statements. It is important to notice that there is not a #for preprocessor statement nor anything similar in C++. If templates are used on C conditional statements (e.g. if, switch), compilers and HLS tools might try to process code blocks that will never be active for the selected configuration. Some C++ compilers and HLS tools are smart enough to avoid processing these blocks of code. However, this feature is not a requirement, so there is not guarantee of these code blocks will not cause any compiler/synthesis error in any tool.

Another important limitation is that array elements or string literals cannot be used to pass values to template parameters.

```
#define NumN(n) ((n)==2 ? N2 : ((n)==1 ? N1 : ((n)==0 ? N0 : -1)))
```

```
Fig. 5 Macro to select the number of neurons of the layer n.
```

Besides, there are also some limitations on the data types which templates can accept. Arrays and user defined types are tricky to add properly. [32] shows how to add this kind of template parameters, but this syntax is complex and it is not sure if every HLS tool will synthesize it correctly.

Furthermore, C++ has no arrays of constants. A C++ const array is a read-only address of an array, but its elements are treated as variables (known only at run time) even if they are addressed by constant indexes.

5.3 ANN Description

The SystemC source code files of the ANN are freely available as one of the benchmkars in the S2Cbench benchmark suite [33]. They were used into a two-tier automatic explorer [22], capable of first find the smallest ANN configuration (for a given maximum error), and then perform the micro-architecture automatic DSE.

Parameters should be carefully decided to be macros or templates, taking into account if they will be used on an #if or they will configure parallel blocks separately.

Authors have solved the absence of array of constants by defining macros like the one shown in Fig. 5. This kind of macros calculate the needed parameters. They can be used in those places where only a constant expression can be used like *e.g.* array length definitions or as template values. However, they lead to excessively long expressions when replacing long arrays.

Layers have been implemented as one template function. Concurrence of layers could be explicitly expressed implementing them as *sc_modules* or *sc_threads*. Implementing them as functions allows HLS tools to extract their concurrency. The layer function template is simply described with two nested for loops.

For regular parallel structures, *i.e.* every block has the same configuration, a for statement can instantiate a configurable number of them. The for iterator is a variable, thus, it can not be used to give values to template parameters or calculate them, but it can be used to give values to inputs, calculate them, or select the outputs (from an array).

A fixed number of parallel separately configured blocks should be instantiated one by one.

Parallel instantiation of separately configured blocks a configurable number of times, has been addressed by authors through two methods. The first one relies on the explorer (or whatever other meta-program that can edit the source code) to write the configured number of calls to the layer function into the source code. The second method uses #if preprocessor statements to hide the calls not needed. This second method does not need an external program, but is limited by the number of layers which can be configured. By implementing a maximum number of calls, the not needed ones can be hidden, but the drawback is that new ones cannot be added without editing the source code.

6. Analysis

The main differences between software and hardware development that affects parameterization are preprocessor features, parameter types, and when decisions are taken. The C++ preprocessor is quite limited, avoiding iterations expressly. Iterations are loops or recursions. Although C++ define templates, as it was shown above, they do not provide always enough flexibility as generics of VHDL does. Function parameters are known at run time. Typically in software, most of the decisions are taken at run time, providing software programs huge flexibility at expenses of slight bigger object code (which most of the times this increase on the code is neglected). On the other hand, on hardware design every optimization that can be done on synthesis time has a great impact.

Summarizing and putting together analysis of VHDL and SystemC ANN descriptions, the main points of improvement for future HLS tools are:

Calculation of constant values during synthesis time: whereas VHDL synthesis tools can calculate constant values even applying functions, SystemC only can use constant expressions. In VHDL, the result of a function which has been called with constant values and do not use internally any run-time data, is treated as well as a constant. Allowing to use this result in such places where only a constant can be used, like length definition of an array, and to perform optimizations. All these computations are skipped from run time, hence no extra logic is inferred. However, C and C++ functions cannot distinguish constant parameters known at compiler-time from variable parameters only known at runtime, so they treated all of them as variable values, carrying out all computations at run time.

Parameter types: VHDL only uses one kind of parameter: the generics. They can be of whatever type, including array and user defined types, and they can be used on if-generate and for-generate statements (which are processed at synthesis time). As opposite, SystemC uses two synthesizable kind of parameters: macros which can interact with the preprocessor but are limited to global scalars or string literals; and templates that can be set independently for different instances of the same element, but they have the following issues:

- They cannot interact with the preprocessor, so they cannot be used to take decisions at synthesis time. If they are used on C conditional statements (if, switch, etc.), compilers and HLS tools may try to process the code blocks that never will be active for the selected configuration, so there is not guarantee of these code blocks will not cause any compiler/synthesis error in any tool.
- Although they can define more parameters types, arrays and user defined types still cannot be guaranteed that every HLS tool will synthesize them correctly. Be-

sides, array elements or string literals cannot be used to give value to template parameters.

Further, C++ does not have constant array of constants, neither anything equivalent to a *const* of an array type in VHDL. A C++ *const* array is a read-only address of an array, but its elements are treated as variables (known only at run time) even if they are addressed by constant indexes.

For-generate: There is not a #for preprocessor statement neither anything similar in C++. Because as it was mentioned above, C/C++ preprocessor avoids iterations expressly. The VHDL for-generate statement is very useful for instantiating parallel modules of the same type without having to copy and paste code. Furthermore, it allows configuring the number of copies. Furthermore in addition, combining it with the generic arrays, each module copy can be set with different parameter configurations.

7. Results of Implementation

On this section are shown results from the implementation of the described IPs with the authors available synthesis tools: ISE, Vivado and CWB. In order to show the flexibility and robustness of our proposed IPs, two different applications are used in this work, described in detailed in the next subsections. The results obtained with our explorer that are already shown on [22] are not presented here.

7.1 Application 1: Classification of Handwritten Digits

This first example application is based on the popular MNIST database [34]. Although in order to make it suitable for embedded computing, the images have been reduced to quarter in both directions by averaging every 8×8 block. So, the inputs to this network are $7 \times 7 = 49$ pixel images.

The outputs of this example are ten values which each one should stand at its maximum to select its assigned digit, and zero in other case. This implies that the output layer of this ANN has a fixed size of 10 neurons.

Table 2 shows how different tools (ISE and Vivado) synthesize the same codes. The parameters for the VHDL IP are Nlayer 2, NbitW 16, NumIn 49, NbitIn 8, NumN {20 10}, 1_type "SP PS", LSbit {12 12}, NbitO {12 8}, and NbitOut 8. The design with "linear" hidden layer has f_type "linear linear" and the design with "siglut" hidden layer has f_type "siglut linear". All of them has been synthesized for a Zynq XC7Z010, which is the SoC in the ZYBO development board. It can be seen that even being the two tools from the same vendor, Vivado uses far more resources than ISE when a "siglut" layer is present. The functionality that both results have is exactly the same, because that is what the language (VHDL) assures. QoR depends on the synthesis tool. In this case the reason is that Vivado does not recognize the sigmoid look-up-table as a ROM, and uses a great amount of LUTs, registers and muxes to implement the same functionality.

Table 3 shows an example of DSE carried out on Vivado HLS for the same configuration than the example of

	"linear" ISE		"linear" Vivado		"siglut" ISE			"siglut" Vivado				
	ANN	SP	PS	ANN	SP	PS	ANN	SP	PS	ANN	SP	PS
Slice LUTs (17600)	2080	1267	804	1305	519	786	2067	1245	812	10780	9979	801
- as Logic (17600)	1104	627	468	969	519	450	1091	605	476	10444	9979	465
- as Memory (6000)	976	640	336	336	0	336	976	640	336	336	0	336
Slice Registers (35200)	871	590	281	1425	628	797	852	570	281	1606	808	797
F7 Muxes (8800)				64	64	0				3316	3284	32
F8 Muxes (4400)				16	16	0				1092	1076	16
BRAM (60)	0	0	0	5	5	0	20	0	0	5	5	0
DSP48E1 (80)	45	20	25	40	20	20	45	20	25	40	20	20

Table 2 Resource utilization of VHDL IP for "linear" and "siglut" hidden layer, synthesized withISE and Vivado.

 Table 3
 Resource utilization of SystemC IP on ZYBO.

	Lat.	Slices	Reg.	LUTs	DSPs	BRAM
unroll complete	140	1577	3758	3600	40	5
unroll factor 10	199	1202	3283	2995	20	15
unroll factor 5	321	1078	3387	2393	10	10
unroll factor 4*	376	949	2841	2437	8	9
unroll factor 2*	1176	930	2803	1679	4	9
VHDL IP (ISE)	68		852	2067	45	20

Table 2 with "siglut" hidden layer. It has been added the solution produced by ISE for the VHDL IP. It can be seen how for the same configuration, HLS can produce different solutions for the trade-off resources vs latency whereas RTL synthesis produces a single optimized fixed architecture. It must be noticed that the VHDL IP uses streaming ports for the ANN data, so input and output buffers are not necesary. On the contrary, the SystemC codes describes such buffers to allow generation of different input and output interfaces. Reading and writing these buffers takes around 60 clock cycles. Hence, the VHDL IP solution is equivalent to the *unroll complete* version. Solutions marked with * may present timing issues.

Figure 6 shows other DSE for another two ANNs. In this case, inputs are $14 \times 14 = 196$ pixel images, and they have 3 layers. The middle layer kind is changed to get extra pareto-optimal architectures.

7.2 Example Application 2: Auto-Encoder

The second application is an auto-encoder for image coding. An auto-encoder is a MLP with as many outputs as inputs, and a narrow hidden layer. When the auto-encoder is trained, inputs are also used as desired outputs. Data is forced to pass through the narrow hidden layer and then reconstructed with minimal error compared to the original data. Thereby, data through the narrow hidden layer is turned to be a compressed representation of the input.

Like other image coding techniques, the full image is divided into blocks, to which the coder is applied individually. In this case, the images are divided into 8×8 blocks, thus, the inputs and output layer have 64 neurons.

This example shows how a little optimized ANN can be fitted onto a low-cost SoC. Figure 7 shows a system implemented on a ZYBO development board [35], which contains an auto-encoder with the following parameters: Nlayer



Fig.6 Latency (horizontal) vs. number of DSPs (vertical). Note: extra latency of 196+10 has been added to VHDL designs to compensate SystemC buffers.



Fig.7 Auto-encoder on ZYBO. The left picture on the monitor is the original, and the right one has been processed by the auto-encoder.

2, NbitW 16, NumIn 64, NbitIn 8, NumN {8 64}, l_type "SP PS", f_type "linear linear", LSbit {12 12}, NbitO {12 8}, and NbitOut 8. And Fig. 8 is the configuration window of the ANN IP in Vivado showing the previous configuration.

520

F Re-customize IP X					
ann_v2.0 (2.0)					
<i>f</i> Documentation 📄 IP L	ocation				
Show disabled ports	Component Name	design_1_ann_0_0			
	C Inputs S AXI	S TDATA WIDTH	32		
	C Outerute M A	<u>د</u> دد			
	Nlayer	2	8		
	Nbitw	16	0		
	Nbitin	8	0		
	Nbitout	8	0		
	Numin	64	0		
	Numn	8 64	0		
	L Type	SP PS	8		
	F Type	linear linear	8		
	Lsbit	12 12	8		
	Nbito	128	8		
		0	K Cancel		

Fig. 8 Configuration window of the ANN IP in Vivado.

8. Conclusion

Table 4 summarizes pros and cons of VHDL and SystemC.

The analysis of the different parameterization mechanisms in VHDL and SystemC has been carried out based on the design of an ANN IP. The resultant source codes from both methodologies are freely available on [29] (VHDL) and [33] (SystemC). As a result of these analysis, we conclude that SystemC's parameterization mechanisms are more complex and less flexible then VHDL's ones.

Although the VHDL description allows some flexibility to generate ANN's with different trade-offs, SystemC can generate a larger variety of different architectures from the same configuration (as it is shown on Fig. 6). Therefore, we believe that the best overall alternative to create re-usable IPs would combine the synthesis possibilities of HLS with a parameterization mechanism as robust and flexible of VHDL's generics. Whereas the increase on the level of abstractions allow to generate different architectures for the same algorithm within the same code, a good parameterization allows to implement more flexible algorithms that avoid to re-write the code for different options.

As traditional RTL synthesis tools can address some of the SystemC HLS weak points, we hope that future HLS tools will address these issues. It is contradictory that lowlevel of abstractions tools can parameterize better complex descriptions like ANNs, but that tools with higher-level of abstraction cannot. Precisely because parameterization becomes more important when the level of abstraction rises.

Regarding the preprocessor issues, one possible solution may be to allow a pre-processing stage during synthesis. As SystemC define three phases for the simulation: elaboration, execution, and cleanup; the elaboration phase could be

Table 4	Summar	y of pros and	l cons of	VHDL and	SystemC.
---------	--------	---------------	-----------	----------	----------

VHDL	Pros - Ad-Hoc optimizations can take advantage of low-level details. - Flexible and efficient pa- rameterization.	Cons - More code lines and de- sign time. - Bigger gap between algo- rithm design and RTL de- sign. - Fixed architecture. Re- design needed for different area vs performance trade- offs
SystemC	 -Less code lines and design time. - Smaller gap between algo- rithm design and HLS de- sign. - Easier and faster DSE. 	 Parameterization mechanisms (inherited from C++) are inefficient and not flexible for hardware synthesis.

used for synthesis-time calculations. The elaboration phase is where every statement is executed prior to the sc_start() call. If synthesis is divided in two phases, the first one can compute the C++ statements in a less restrictive way to get a set of values known at the end of this phase, let us call them post-elaboration constants, and to program the system structure instantiating modules using if and for statements. Note that this may imply to allow the creation of dynamic objects during this phase. This also open the possibility of using constructor parameters.

Acknowledgments

This work was partially supported by the Consejo Social de la Universidad Politécnica de Madrid; and the Universidad Politécnica de Madrid under the grant RR01/2015 (Programa Propio).

References

- O. Pell and O. Mencer, "Surviving the end of frequency scaling with reconfigurable dataflow computing," SIGARCH Comput. Archit. News, New York, NY, USA, pp.60–65, ACM, Dec. 2011.
- [2] "IEEE standard for standard systemc language reference manual," Jan. 2012.
- [3] "Systemc synthesizable subset version 1.4.7," https://accellera.org/ images/downloads/standards/systemc/SystemC_Synthesis_Subset_1_ 4_7.pdf, 2016.
- [4] J. Misra and I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress," Neurocomputing, vol.74, no.1-3, pp.239–255, Dec. 2010. Artificial Brains.
- [5] A.R. Omondi and J.C. Rajapakse, eds., FPGA implementations of neural networks, Springer, 2006.
- [6] J. Zhu and P. Sutton, "FPGA implementations of neural networks: A survey of a decade of progress," in Field Programmable Logic and Application, pp.1062–1066, Springer Berlin Heidelberg, 2003.
- [7] M. Nigri, P. Treleaven, and M. Vellasco, "Silicon compilation of neural networks," CompEuro '91. Advanced Computer Technology, Reliable Systems and Applications, 5th Annual European Computer Conference, Proceedings., pp.541–546, May 1991.
- [8] E.M. Ortigosa, A. Cańas, E. Ros, P.M. Ortigosa, S. Mota, and J. Díaz, "Hardware description of multi-layer perceptrons with different abstraction levels," Microprocessors and Microsystems, vol.30, no.7, pp.435–444, Nov. 2006.
- [9] G. Smaragdos, S. Isaza, M.F. van Eijk, I. Sourdis, and C. Strydis,

"Fpga-based biophysically-meaningful modeling of olivocerebellar neurons," Proc. 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays, FPGA '14, pp.89–98, ACM, 2014.

- [10] N. Farrugia, F. Mamalet, S. Roux, F. Yang, and M. Paindavoine, "A parallel face detection system implemented on fpga," 2007 IEEE International Symposium on Circuits and Systems, pp.3704–3707, May 2007.
- [11] C. Torres-Huitzil, B. Girau, and A. Gauffriau, "Hardware/software codesign for embedded implementation of neural networks," Proc. 3rd Int. Conf. Reconfigurable Computing: Architectures, Tools and Applications, ARC'07, Berlin, Heidelberg, pp.167–178, Springer-Verlag, 2007.
- [12] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.s. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," Proc. 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '16, New York, NY, USA, pp.16–25, ACM, 2016.
- [13] M. Bettoni, G. Urgese, Y. Kobayashi, E. Macii, and A. Acquaviva, "A convolutional neural network fully implemented on fpga for embedded platforms," 2017 New Generation of CAS (NGCAS), pp.49– 52, Sept. 2017.
- [14] G. Lacey, G.W. Taylor, and S. Areibi, "Deep learning on fpgas: Past, present, and future," arXiv preprint arXiv:1602.04283, 2016.
- [15] D. Lettnin, A. Braun, M. Bodgan, J. Gerlach, and W. Rosenstiel, "Synthesis of embedded SystemC design: A case study of digital neural networks," Proceedings Design, Automation and Test in Europe Conference and Exhibition, vol.3, pp.248–253, IEEE Computer Society, 2004.
- [16] S. Chtourou and O. Hammami, "Design space exploration of SystemC SOM implementation," Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on, pp.159–169, Dec. 2005.
- [17] A. Gomperts, A. Ukil, and F. Zurfluh, "Development and implementation of parameterized fpga-based general purpose neural networks for online applications," IEEE Trans. Ind. Informat., vol.7, no.1, pp.78–89, Feb. 2011.
- [18] M. Baharani, H. Noori, M. Aliasgari, and Z. Navabi, "High-level design space exploration of locally linear neuro-fuzzy models for embedded systems," Fuzzy Sets and Systems, vol.253, no.Supplement C, pp.44–63, 2014. Theme: Fuzzy Modeling and Clustering.
- [19] M. van Eijk, C. Galuzzi, A. Zjajo, G. Smaragdos, C. Strydis, and R. van Leuken, "Esl design of customizable real-time neuron networks," 2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) Proceedings, pp.671–674, Oct. 2014.
- [20] E.J. Kreinar, "RFNoC neural network library using Vivado HLS," Proc. GNU Radio Conference, vol.2, no.1, p.7, 2017.
- [21] "revision zone," https://www.xilinx.com/products/design-tools/ embedded-vision-zone.html.
- [22] B. Carrion Schafer, D. Aledo, and F. Moreno, "Application specific behavioral synthesis design space exploration: Artificial neural networks. a case study," 2017 Euromicro Conference on Digital System Design (DSD), pp.129–136, Aug. 2017.
- [23] M.K. Hamdan and D. Rover, "Vhdl generator for a high performance convolutional neural network fpga-based accelerator," ReConFig, pp.1–7, Dec. 2017.
- [24] H. Zeng, C. Zhang, and V. Prasanna, "Fast generation of high throughput customized deep learning accelerators using fpgas," Re-ConFig, pp.1–7, Dec. 2017.
- [25] H. Sharma, J. Park, D. Mahajan, E. Amaro, J.K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp.1–12, Oct. 2016.
- [26] S.I. Venieris and C.S. Bouganis, "fpgaconvnet: A framework for mapping convolutional neural networks on fpgas," 2016 IEEE 24th

Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp.40–47, May 2016.

- [27] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates," 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp.152–159, April 2017.
- [28] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning internal representations by error propagation," Tech. Rep., DTIC Document, 1985.
- [29] D. Aledo and F. Moreno, "ANN," https://opencores.org/project/ artificial_neural_network, June 2016.
- [30] D.D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, Embedded system design: modeling, synthesis and verification, Springer Science & Business Media, 2009.
- [31] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," IEEE Design Test of Computers, vol.26, no.4, pp.18–25, July 2009.
- [32] http://en.cppreference.com/w/cpp/language/template_parameters.
- [33] The Hong Kong Polytechnic University, DARClab, "S2CBench v.1.1," 2015. http://sourceforge.net/projects/s2cbench/.
- [34] Y. LeCun, C. Cortes, and C.J.C. Burges, "The MNIST database of handwritten digits," http://yann.lecun.com/exdb/mnist.
- [35] "Zybo," https://reference.digilentinc.com/reference/programmablelogic/zybo/start.



David Aledo received the BEng degree in industrial engineering (electrical engineering) and the MSc degree in industrial electronics from Universidad Politécnica de Madrid (UPM), in 2011 and 2013 respectively. He is Ph.D. student and researcher in Centro de Electrónica Industrial of UPM. His research interests include digital electronic design on FPGAs, digital signal processing, high-level synthesis, artificial neural networks, and wireless sensor networks.



Benjamin Carrion Schafer completed his Ph.D. at the University of Birmingham, U.K. in 2002. He is professor at Department of Electrical and Computer Engineering, University of Texas at Dallas, USA. His research interests include reconfigurable computing, thermal-aware VLSI design and high-level synthesis. He was also a member of Accellera's SystemC synthesizable user group committee, leading the effort to standardize a synthesizable subset of SystemC.



Félix Moreno was born in Valladolid, Spain, in 1959. He received the MSc and Ph.D. degrees in telecommunication engineering from Universidad Politécnica de Madrid (UPM), in 1986 and 1993, respectively. Currently, he is Associate Professor of Electronics at UPM. His research interests are focused on evolvable hardware, high-performance reconfigurable and adaptive systems, hardware embedded intelligent architectures, and digital signal processing systems.